(Presented at National Instruments User Symposium, July 26-28, 1995, Austin, Texas)

VISUAL PROGRAMMING IMPROVES COMMUNICATION AMONG THE CUSTOMER, DEVELOPER, AND COMPUTER

by

Christopher Hartsough, M.S. and Ed Baroth, Ph.D.
Measurement Technology Center (MTC)
Jet Propulsion Laboratory,
California Institute of Technology

ABSTRACT

This paper reports direct experience with commercial, widely used visual programming languages. In the environment of test and measurement of flight equipment, visual programming currently provides productivity improvements of from four to ten times compared to conventional text-based programming. The most dramatic gains in productivity are attributed to the cornmu nication among the customer, developer, and computer that are facilitated by the visual syntax of the tools. The customer understands enough of the visual diagram to make suggestions and modifications in the forma] statement of the problem. Using these tools blurs the requirements, design, and coding phases into a single activity, usually implemented interactively with the customer and developer together at the computer. The authors have not found a text-based environment where these conditions hold. This paper discusses the expansion of visual languages and tools into more of the process than is currently 'standard practice.' The improvements in productivity demonstrated from user-understandable programming leads to proposing the expansion of visual programming into other engineering disciplines.

INTRODUCTION

The Measurement Technology Center (MTC) evaluates commercial test and measurement hardware and software products that are then made available to experimenters at the Jet Propulsion Laboratory. The MTC specifically configures, delivers and supports turn-key measurement systems including software, sensors, signal conditioning, and data acquisition, analysis, display, simulation and control capabilities. [1,2]

Visual programming tools, specifically LabVIEW and HP VEE, are frequently used to configure these systems. Visual programming tools that control off-the-shelf interface cards have been the most important factor in productivity improvements of from four to ten times compared to conventional text-based programming.[3,4] The gains in productivity are attributed to the communication among the customer, developer, and computer that are facilitated by the visual syntax of the tools, thereby reducing time and cost of configuration and modification.

Our evidence shows that these tools facilitate communications because they provide a common expression that can be read by our customers, the developer, and the computer. There are different details of each syntax that facilitate this communication, but the details are unimportant: what is important is the transformation of requirements from a statement to a d ynarnic conversation that results in system components as a natural outcome of the process.

The visually based syntax is the key factor in the acceptance of visual programming by our customers. Our experience is that the development paradigm of a 'Requirements' definition followed by an implementation phase is obsolete in our test and measurement environment. The process now more closely represents rapid applications development (RAD)[5], anti eliminates a separate implementation phase because, frequently, when the requirements definition has been completed, so has the system. Traditionally, the

Requirements definition is part of the communications chain that ultimately ends with the developer coding at the computer. Visual programming can eliminate the chain of communications between customer, developer, and computer because coding usually is implemented interactively with the customer and developer together at the computer as a single conversation.

Most readers have had similar experiences in quickly and successfully building data acquisition systems using a visual programming language/environment. If your organization is like the MTC, you build such systems with regularity and under schedule pressure. In contrast to suggesting improvements in the areas where visual languages are currently most active and effective, or with general purpose programming language issues, this paper focuses on the test /characterizat ion process. This paper discusses the expansion of visual languages and tools into more of the process than is currently 'standard practice.'

A context is needed before boldly going into areas where LabVIEW* has never gone before. In previous papers,[6,7] the authors have presented a model of conversation, language, and communication as the underlying basis for the most dramatic productivity improvements that the M TC has achieved. We have found that the visual diagram is a natural and easy representation that the users understand enough to make suggestions and modifications in the formal statement of their problem. We have also found that in the visual environment, implementation (which includes design plus programming plus test) is fast enough for conversation to occur in the language of the diagram. We have not found a text-based environment where these conditions hold. The productivity improvement we've witnessed, resulting from programming that is understandable to the user, suggests the expansion of visual programming to other engineering disciplines.

CONTEXT FOR DEVELOPMENT

Most people using a visual programming environment fall into one of the following groups:

- Engineers with a need to test something they've designed,
- Test engineers who are tasked to test something somebody else designed, or
- An implementation specialist — developer (or perhaps a 3rd party) — supporting one of the above.

In each of the above cases, there needs to be a conversation with the visual programming system that facilitates communication. "Conversation" is the appropriate term: saying something in the visual language to the visual programming system (computer), and the computer communicates with you. When the conversation is successful, the program works correctly.

When a developer is supporting an engineer or user, achieving end-to-end communication gets a bit more complex. There is another talker/listener in the conversation and a possible rub: what if the language used in the conversation between the customer and the developer is different from the language used between the developer and the computer? Anyone who has tried to fully commu nicate through a translator can answer that question. Anyone who has acted as a translator can attest to how hard it is to cross the cultural barrier created by two languages. Programming in C or BASIC based on English requirements is a translation. When the language of the conversation is the same for all parties, as it is for visual but not for text-based programming, then the communication barrier is reduced. This is a real advantage that visual programming brings to the development process. It is what enables a conversation-based paradigm.

---

*There really are other visual programming environments, at least one focused on test, On the basis of on who is attending this conference the prevalent tool discussed will be LabVIEW, Our remarks, however, are not LabVIEW specific.

When a developer and user sit together at the computer, all three have a communication in a visual development context and the language barriers are reduced. What occurs is a conversation between two groups (customer to developer and developer to computer) who both nearly speak one language. While there are side conversations in native tongue, the intent of the side conservation is to formulate a better expression in the joint language, the computer language. On the basis of our experience, this process is completely different from a linear translation from A through B to C. It is where the conversation among customer, developer, and computer takes place in a visual language context that the highest productivity improvements have been achieved in the MTC.

This increase in productivity with engineers and scientists is largely due to visual programming specifically based on the data flow design (DFD) and flow-charting.[8] These structural syntaxes are comfortable to engineers and scientists, which are our customers and users. Most have limited programming experience with either visual or text-based code. Most, if not all, understand data flow diagrams, so it becomes a natural mode to use as the language of discussion. We have consistently found users with little or no experience in LabVIEW or VEE could 'understand' the process, if not the details, of the program. By programming together at the terminal, customers can follow the data flow diagrams enough to make suggestions or corrections in the flow of the code. It is difficult to imagine a similar situation using text-based code, where someone with little or no understanding of 'C' could correct a programmer's syntax or flow. Actually, it is difficult to imagine anyone 'watching' someone else program using text-based code at all.

The problem with other visual tools in common use today (e.g., Visual Basic, Visual C, Visual 'x') is that the program **itself** is text-based and not visual. While the output is certainly graphical and can be very effective and visual, the development remains text-based and therefore is not suitable for the conversation paradigm. In the MTC there are those who can develop a program as fast in Visual 'x' as in LabVIEW when the specifications are essentially complete as in a re-implementation of existing software. When the problem involves requirements discovery, design, implementation or test, then the Visual 'x' languages cannot support the human information transfer rates needed to sustain a communication. The text-based forms, at least so far, cannot be read or understood by the customer as they are being written; therefore the whole conversation paradigm breaks down. This is what separates visual programming environments such as LabVIEW from text-based (including object-oriented) environments.

The productivity improvements available in conversation have not gone unnoticed by visual language developers: notably, the development of application specific graphic programming tools.[9] While this is not precisely the same model we are advocating, the conversation element is, in our opinion, the key to the success. In the cited work, the subject and object of the conversation is a graphical language itself, which is somewhat more complex than the conversations in the MTC. Additional remarks regarding visual programming studies are given in the Appendix.

EXTENDING THE PARADIGM

The MTC has already demonstrated[10,11] that LabVIEW can be successfully used in areas not originally intended (e.g., telemetry, simulation, pure data analysis) and provide the same productivity improvements as when used in 'typical' applications such as data acquisition, It is our belief that the productivity improvements are mainly due to the conversation that occurs when the developer and user are programming together at the computer. It seems likely that these productivity improvements can be extended to other elements of the development process.

Making LabVIEW or any visual tool better for current users certainly can't hurt, but that's not the purpose of this paper. Making the tool accessible to a wider community is. If we can extend to a wider group of participants the same increases in productivity and integration of activities that we have already provided to the test and measurement community, the resulting productivity will be more than the sum of its parts. If we take the communications model as a base, we can expand naturally to the next ring of communications partners. Figure 1 shows one potential set of new conversation partners.

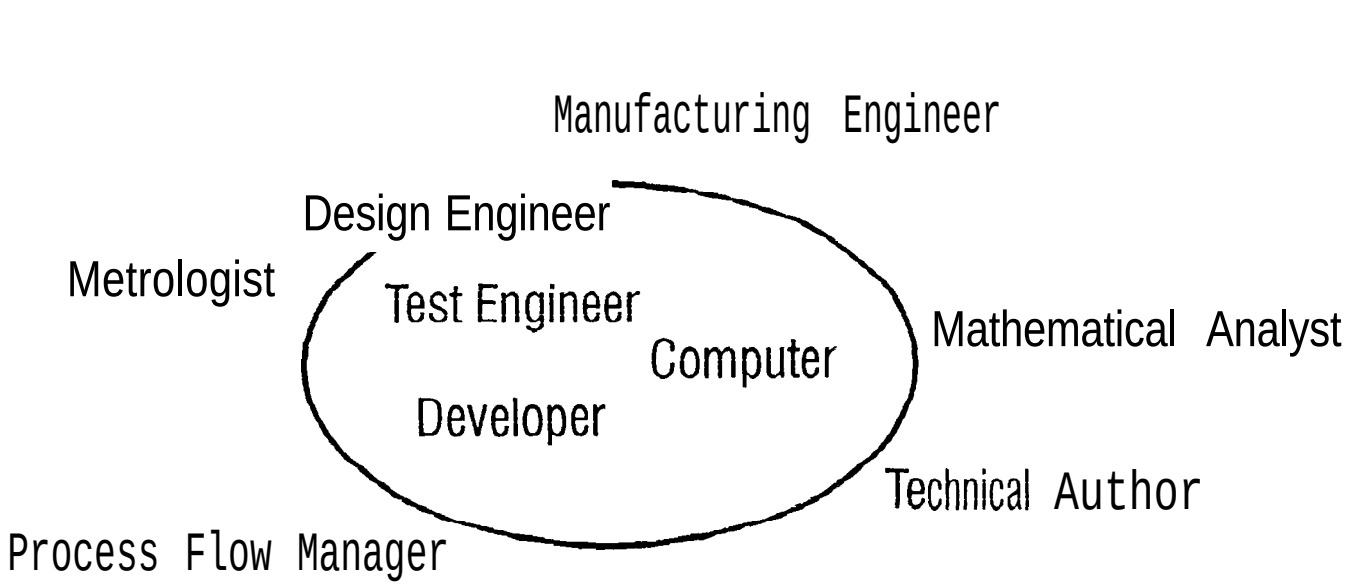

Figure 1. Next level of conversation partners

As we look at each new possible member of the communication group, wc notice that there are tools they already use. It is very tempting to build a 'bridge' that allows commands in the extension tool to be issued from the visual environment, For example, it is fairly straightforward to create a LabVIEW subVI that directly interfaces to Hi-Q and passes text-strings that are Hi-Q commands. This class of bridge is helpful, but demanding. The reason is language. You now have added another language, in this example that of Hi-Q commands, to a conversation that was already in English and LabVIEW. It is as if one is translating English to French with a bit of precise Swahili thrown in as well. It is no wonder that these bridges aren't wildly popular. When a person can use this type of bridge effectively, the are often (justifiably) proud of their accomplishment.

Often the use of the bridge and the additional language complexity is not productive and it is more effective to create an interface (e.g., a file or pipe) and do the next phase of the work in a new and separate language context. This is not what we think the future should look like. Applications should not require a major shift in the language of expression.

> ... In fact the great success of the Macintosh can be attributed to a large degree to the efforts of the early Apple "evangelists" to encourage applications developers to use a common design vocabulary. They facilitated this by publishing explicit guidelines[12], by providing tools for all of the standard elements . . . and by working directly with developers. They convinced the developers that they would gain more from promoting the popularity of the Mac platform by making it seem easy to use through uniformity, rather than through having minor differences (improvements) unique to their interface.[13]

In the Mac example, they limited the vocabulary to 'look and feel.' The authors feel that this philosophy needs to be extended into the functional or semantic content as well. in the MTC environment, this includes test requirements, design for testability, metrology, data acquisition and analysis, data presentation, data archiving, and database, at a minimum. If LabVIEW (for example) had icons that access Hi-Q functionality from within LabVIEW, in LabVIEW syntax, one could take advantage of the power of Hi-Q without having to learn its command structure. This kind of functionality would directly support the role of the mathematical analyst. This is true for other elements in the development process as well: LabVIEW could allow all the capabilities and functionality associated with the roles shown in Figure 1 to be accessed without having to learn a new tool and language for each one.

This idea of conversation among disciplines exists in the commercial market, e.g., Spectragraphics Corp. offers a 'Team Solution' package that allows X-Windows displays to be viewed simultaneously by team members. In fact, they sub-title their team exchange product "Better I'roduct Conversations. " In the entertainment world, Emotion (company and product) offers a product that allows communication based on a time code in a multi-media production. This product allows comments, alternatives, etc. to be exchanged by a team working on a single product. What differentiates these products from what we're suggesting is that these products use the computer to facilitate the communication, not as a participant in the conversation.

It is felt that productivity improvement will be best served by having conversations with many participants of differing responsibilities, including the computer. In these conversations, each party must have full access to the domain specific functionality that they are there to contribute. To facilitate these conversations, languages arc needed that don't require the participants' attention. They should not need to worry about: "What syntax do we use here?" This argues AGAINST overly rich syntax(s) that do things the 'best' way in each case. It argues for consistent and powerful constructs that can be universally applied and exploits intuition and simplicity. As more tools are integrated, the importance of, in popular terminology, 'seamless integration' becomes clear, It is at the seams that requires one to stop and solve yet another pesky syntax problem. (The authors suspect that the consistency of language constructs across application domains is the essence of 'seamless integration' of applications,) To the extent that many parties can converse with the computer about the problem smoothly and naturally, we can extend the conservation style and reap the attendant productivity increases.

This point deserves elaboration. It is not just the formalisms of the language that need to be accounted for, there also idiomatic issues. In this paper we are using the term idiom as it is synonymous with "accepted phrase, " "expression," and "idiosyncrasy of speech, " Idioms in the language arise and while the idioms are not proscribed by the language, but are supported by it. Attempting to do things in other ways do not work conveniently. It is through these idioms that the efficiency of the language is fully utilized.

The development of idiomatic forms is a cultural phenomenon, both macro and micro. With natural languages, idiomatic expression is the mechanism of language extension. In formal languages, the idioms of the users' culture are the source material for language extension. With formal languages there is often a tug-of-war between practical cultures that want 'to get something done' and academic cultures that want 'to get it done correctly.' At a micro level, cultural idiom often takes the form of "Coding Standards" both formal and informal. In FORTRAN shops, for example, it is often the idiom that format statements have labels over 1000 and/or all subroutines end with a "999 CONTINUE" followed by the RETURN.

Idiom is also an expert phenomenon. As the user of a language becomes more expert, the structures used, what Petre[14] and others call Secondary Notation, become regular and informative. What develops is an idiomatic form of *expression beyond the formal definition of the language* that assists in carrying the information content of the communication. While Petre is primarily concerned with 'cold reading' of material rather than the construction of the material, the observations about how experts read and write point to training needs for developers.

What is most important about the idiomatic part of the language culture is that it simplifies communication, the 'what and why' of common structures at a glanc e. Jargon, when it is not being used to exclude, is another form of idiomatic and fast communication. In formal languages, we frequently use naming conventions as jargon to easily predict the role/function of a language element. Most of us would consider a sine function called "aardvark" an error, although the computer couldn't care less.

The discussion about idiom is going somewhere. The "where" is: NOT ONLY must domain specific tools be integrated into the formal language of a system, they must be integrated into the predominant idiom of that language! If this is not done, it won't be as bad as a Frenchman talking to a German; but it will be like upstate Maine talking to back-country Alabama. If there becomes a seamless integration and a conversational model for development, as we advocate, just what does the developer bring to the party?

ROLE OF THE DEVELOPER

Why do we need a developer? For several things, actually. Even if there are no seams, there may be wrinkles, and the developer brings the iron. There are several useful, and one critical, things that the developer brings:

- Formal language skills.

  The developer is the 'author' of the diagram, and the customer is the 'editor.' From experience, we have found that the customer can read and edit the diagram as it is being produced long before the customer could realistically create a diagram (program) on their own.

- Development and operations environment knowledge.

  For the foreseeable future, there will be an operating system underlying the graphical environment and its needs must be tended to. When the program is running, there are other operating systems issues that must likewise be attended.

- Data integrity and/or database expertise.

  Testing in the modern setting implies data logging and historical records in some form. Responsibility for these areas is jointly held, AND developers should hold themselves specific responsible for this area of the system.

- Guiding the conversation for development.

  The developer is in the best position to understand what has not been specified (the computer knows too, it's just a bit rude). The responsibility of managing the conversation for completeness is the developers.

- STRUCTURE

  This is the most critical contribution of the developer toward the overall success of the project!

There are several arena's that need structure. Clearly the program needs some structure. 'Spaghetti' is a structure, or lack there of, that has earned a bad name in the software world. It deserves it. (A different form of pasta design has a following in the UNIX world: Shell scripts.) If the problem is small, and the life of the software is on the order of 30 days, just throwing it together AND then throwing it away (!) is not necessarily a bad idea. If the problem has any size OR life span, then structure is needed. There is plenty of choice: (in no particular order)

- HIPO (Hierarchical Input Process Output)
- Data Driven
- Event Driven
- Decision Tables
- Structured Design
- Data Flow
- Object-Oriented
- more; lots more . . .

It's beyond the scope of this paper to delve into these, or any other, design structures and their methods. Some of the above structuring are out of vogue just now. Some deserve to be. What we have found is that these methods all have their limitations and strengths. What is important here is that the system will get a structure, and the key questions are: "Will the structure be designed or defaulted?" and "Will the structure support or hinder the processes of system development, system operation, and system maintenance?" If the developers don't know what these techniques are, it will be hard for them to select the one(s) that is best. For better or worse, it will be the developer that spans the full spectrum of conversations and it will be the developer that has the only opportunist y to supply system structure.

What else needs structuring? The diagram itself. The structure of the drawing itself is critical to the conversation. The discussion of secondary notation (above and Petre[14]) addresses the importance of this structure. The structuring of the conversation has been noted. What we are emphasizing is perhaps an element of design, where 'design' is meant in its broader context than engineers are accustomed to applying to our own work,

SUMMARY

The most important advantage the MTC has found in using visual programming is the support for communications among the customer, developer, and hardware that visual programming enables. Without the visual component, the support for communications is not present.

The visual component is the ability to graphically communicate the state of execution of a system to the customer. This capability to see what the 'code' is doing directly is of inestimable value. The graphics description of the system without the animation would be not much more than a CASE tool with a code generator; with the animation, the boundaries between requirements, design, development, and test appear to collapse. Seamless movement from one activity focus to another makes the development different in kind, not degree. This is because we can sustain the communication among the customer, developer, and computer. If there are substantial time lags in changing tools, (e. g., conventional debuggers or third party applications) the conversational environment breaks down.

Having additional capability (data analysis, visualization, database, etc.) in the same iconic format extends the language and increases the communication. This expands the possibilities into other engineering disciplines. Access to other tools is, of course, better than nothing; but if the language used to control the third party tool is different from the host language, the advantage is severely diminished. It would be as if you took your car in for repair and had to describe engine problems in English, suspension in French, breaking in German, and so on. It's better than not getting your car repaired, but far from ideal.

Given where we are, where we generally train people in the formal language skills primarily, and given where we need to be, where developers are pivotal members of the entire process, we need education. We need to provide education in thinking processes in addition to specific skills. This is a path less traveled but not uncharted. We need to educate ourselves in structuring, expression and conversation. As suppliers, we need to provide tools that promote expression and conversation across technical disciplines. This path leads to increased productivity across technical disciplines and an overall improvement of the process through redefining the user base.

ACKNOWLEDGMENTS

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

1 E. C. Baroth, D. J. Clark, and R. W. Losey, "Acquisition, Analysis, Control, and Visualization of Data Using Personal Computers and a Graphical-Based Programming Language," Conference *Proceedings of American Society of Engineering Educators (ASEE)*, Toledo, Ohio, June 21-25, 1992, pp. 1447-1453.

2 E. C. Baroth, D. J. Clark, and R. W. Losey, "An Adaptive Structure Data Acquisition System using a Graphical-Based Programming Language, " *Conference Proceedings of Fourth AIAA/Air Force/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization*, Cleveland, Ohio, September 21-23, 1992,

3 G. Wells and E. C. Baroth, "Telemetry Monitoring and Display using LabVIEW," *Conference Proceedings of National Instruments User Symposium*, Austin, Texas, March 28-30, 1993.

4 D. Breeman, "Jet Propulsion Lab Aids in Space Craft Project," Scientific Computing and Automation, November, 1993, pp. 26-28.

5 E. Yourdon, *Decline and Fall of the American Programmer*, Yourdon Press, Prentice Hall Inc., Englewood Cliffs, 1992.

6 E. C. Baroth and C. Hartsough, "Experience Report: Visual Programming in the Real World," *Visual Object Oriented Programming*, edited by M. M. Burnett, A. Goldberg & T. G. Lewis, Manning Publications, Prentice Hall, 1995, pp. 21-42.

7 E. C. Baroth and C. Hartsough, "Visual Programming as a Means of Communication in the Measurement System Development Process," SciTech Journal, Vol. 5, No. 5, pp. 17-20.

8 J. Kodosky, J. MacCrisken, and G. Rymar, "Visual Programming Using St ruct u red Data Flow," Proceedings of the 1991 IEEE Workshop on Visual Languages, Kobe, Japan, October 8-11, 1991, pp. 3,4-39.

9 A. Repenning, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," Computer (IEEE), March, 1995, pp. 17-25.

10 G. Wells and E. C. Baroth, "Using Visual Programming to Simulate, Test, and Display a Telemetry Stream," MacSciTech's SEAM '95 Conference, San Francisco, California, January 8-9, 1995.

11 F. Razo and J. McGregor, "Using A Visual Language to Process Space Shuttle Telemetry Data," American Society of Engineering Educators (ASEE), Anaheim, California, June 19-23, 1995.

12 Apple Computer, *Human Interface Guidelines: The Apple Desktop Interface*, Addison-Wesley, Reading, Mass., 1987.

13 T. Winograd, 'R-em Programming Environments to Environments for Designing,' Communications of the ACM, June 1995, Vol. 38, No. 6, pp. 71.

14 M. Petre, "Why Looking Isn't Always Seeing," Communications of the ACM, June 1995, Vol. 38, No. 6, pp. 33-44.

15 T. R. G. Green, M. Petre, and R. K. E. Bellamy, "Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture," Fourth Workshop on Empirical Studies of Programmers, New Brunswick, New Jersey, December 7-9, 1991, pp. 121-146.

16 T. G. Moher, D. C. Mak, B. Blumenthal, and L. M. Leventhal, "Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets," Fifth Workshop on Empirical Studies of Programmers, Palo Alto, California, December, 1993.

17 R. K. Pandey and M. M. Burnett, "Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Imperical Study," Oregon State University, Department of Computer Science, 93-60-08.

APPENDIX - REMARKS REGARDING VISUAL PROGRAMMING STUDIES

There have been a few studies comparing visual with other types of programming, and those that do exist have focused on aspects that do not seem to correspond with our use of visual programming. The study by Green et al.[15] compared readability of textual and graphical programming (LabVIEW). Their clear overall result was that graphical programs took longer to understand than textual ones. The study by Moher et al.[16] essentially duplicated the study by Green et al. but compared petri-net representations with textual program representations. They duplicated some of the earlier results, but did find areas where the petri-net representation was more well suited, albeit with reservations. Additional studies[14] tend to confirm the previous work.

The cited studies have focused on experienced users of visual or textual code. No study measured the time to create or modify the programs. It is in these areas, that of user (not programmer) experience and time to create and modify programs, that we find advantages in visual over textual programming in our real world. In addition, these studies used only static visual representations, whereas in real world systems, customers and developers get to interact with the program while trying to understand it. Most importantly, none of these studies considers the conversation aspect available using visual programming.

The study by Pandey and Burnett[17] did compare time, ease and errors in constructing code using visual and text-based languages. The programs chosen were on the level of 'homework' type tasks, certainly not real world problems, but even at that level they did find evidence that matrix and vector manipulation programs were more easily constructed and had fewer errors using visual programming.

Using visual programming at this last stage of the coding process, however, removes much of the advantages we've seen. Once specifications are determined, it simply becomes a race to see who can type faster or who has access to more or better libraries of code or icons. The real benefit we find in using visual programming is the flexibility in the design process, before requirements have been determined. The user-programmer-computer communication is substantially improved due to the speed at which modifications can be made because of the conversation allowed by the visual syntax.

None of the existing studies have addressed the key issue in this paper, i.e., ability of users, because they can 'read' the visual code, to make suggestions and corrections in the program as it is being written. None have dealt with the ability of visual vs. text-based programming to solve real world problems, i.e., determine specifications, create, modify code and user interfaces as well as train inexperienced users to both operate and modify systems. Studies need to be done which allow creativity in constructing, testing and modifying models using visual and textual programming.